CSCI-GA 2590: Natural Language Processing
Hidden Markov Models

Name
NYU ID

Acknowledgement:
This homework is adapted from Jason Eisner's HMM homework from the NLP course at JHU.

Collaborators:
By turning in this assignment, I agree by the honor code of the College of Arts and Science at New York University and declare that all of this is my own work.

In this homework, you will build an English POS tagger using the given data. You just need to submit your code. No written part is required for this homework.

1 Data

Datasets. There are two datasets under `data`.

1. **Ice cream climatology dataset (ic).** This is a synthetic toy dataset for you to test and debug your code. We strongly encourage you to start with this easy dataset. The observations are the number of ice cream cones eaten by Jason each day. The hidden states are the whether each day (`C` for cold and `H` for hot). You will use `ictrain` for training and `ictest` for testing. However, this dataset is mainly for debugging purposes and you don't need to submit any code for it.
2. **English POS tagging dataset (en).** This dataset consists of word sequences and the POS tag for each word. We use coarse English tags in this homework, i.e. only the first character of the original tag is used, e.g., different types of nouns (`NN`, `NNP`, `NNS`) are reduced to simply nouns (`N`). The tagset is described in Table 1. You will develop your model using `entrain` and `endev`. We will run you code for training and testing on a hidden test set.

File format. Each line has a single word/tag pair separated by the `/` character. Punctuation marks count as words. The special word `###` is used for sentence boundaries, and is always tagged with `###`. You can think of it as the special start symbol `*` and the stop symbol `STOP` too. When you generate `###` that represents a decision to end a sentence (so `###` = `STOP`). When you then generate the next tag, conditioning on `###` as the previous tag means that you're at the beginning of a new sentence (so `###` = `STOP`).

2 Evaluation

We will evaluate your program based on the model's accuracy and the training speed. However, we suggest you print additional metrics during model debugging and development.

accuracy percentage of test tokens that received the corret tag, *excluding* the sentence boundary markers `###`.

known word accuracy consider only tokens that appear in the training data.

`C`	Coordinating conjunction or Cardinal number
`D`	Determiner
`E`	Existential there
`F`	Foreign word
`I`	Preposition or subordinating conjunction
`J`	Adjective
`L`	List item marker (a., b., c., ...) (rare)
`M`	Modal (could, would, must, can, might...)
`N`	Noun
`P`	Pronoun or Possessive ending ('s) or Predeterminer
`R`	Adverb or Particle
`S`	Symbol, mathematical (rare)
`T`	The word to
`U`	Interjection (rare)
`V`	Verb
`W`	wh-word (question word)
`###`	Boundary between sentences , Comma
`.`	Period
`:`	Colon, semicolon, or dash
`-`	Parenthesis
`‘`	Open quotation mark
`’`	Close quotation mark
`$`	Currency symbol

Table 1: Tags in the `en` dataset.

novel word accuracy consider only tokens that do *not* appear in the training data, in which case the model can only use the context to predict the tag.

3 Implementing the Tagger

Dependencies. Your code should run with `python3`. You can use any native python functions and the `numpy` package. Please do not import additional libraries which will break the auto-grader.

Input/output. Implement a tagger that runs as follows (on the `ic` dataset):

```
python tag.py --train ictrain --test ictest
```

It reads in the training and test files, learns a HMM tagger, and writes a prediction file called `test-output`. `test-output` should use the same format as the test file (i.e. each line has a single word/tag pair). We will run your code to train a model and evaluate the predictions using `test-output`. (Note that during grading we may randomize the tags in the test file, so please do not simply write the test file the program reads in.)

Key steps. You are free to implement the tagger however you want. Here are the suggested steps:

1. Read the training data and store the word-tag and tag-tag counts.
2. Compute the emission and transition probabilities in HMM.
3. Read the test data and compute the tag sequence that maximizes $p(\text{words}, \text{tags})$ given by the HMM.
4. Compute and print the accuracy on the test data and other metrics or statistics.

Make sure to do all computation in the log space.

Check your implementation on the `ic` dataset. Implement a vanilla HMM tagger using unsmoothed counts and Viterbi decoding covered in lecture 6. The program `tag.py` should produce an accuracy of 87.88% or 90.91%. There are two accuracies because of arbitrary tie breaking during decoding. It may seem to be a problem on this toy data, but it's very rare to have two sequences with the same probabilities in realistic datasets. Therefore, breaking ties arbitrarily is common in practice. Next, let's improve the tagger to make it work on the `en` dataset.

Smoothed probabilities. To make the tagger work on the real data, the first thing you need to implement is smoothed probabilities. It's common to encounter unseen word-tag and tag-tag pairs at test time. Without smoothing, you might see `nan` warnings during inference. For example, you can use simple add-1 smoothing.

Pruning. Now, if you try to run the tagger on the `en` dataset, you may find that it's pretty slow even with our reduced coarse tagset. You can speed up the tagger by ignoring word-tag pairs that do not appear in the training set. You can derive a tag dictionary from the training set that records the set of allowed tags for each word, and only consider allowed tags for each word during viterbi decoding. For unseen words, all tags are allowed. With pruning, we may miss the actual best tag sequence, but it would significantly improve the decoding speed.

Unigram tagger. You can easily adapt the tagger to use unigram transtion probabilities $p(y_i)$, i.e. the joint probability is

$$p(x, y) = \prod_{i=1}^{n} p(x_i \mid y_i) p(y_i) .$$

This tagger is very fast because it essentially predicts the tag that maximizes $p(y_i \mid x_i)$ for each word separately. On `endev`, this baseline tagger gets an accuracy of 91.59% with add-1 smoothing.

Improving the tagger. The baseline unigram tagger is actually quite accurate with an accuracy of 91.59%. You bigram tagger should at least beat this baseline. Better accuracy or speed will get more credits. Here are some possible improvements:

- In the baseline we used simple add-λ smoothing where $\lambda = 1$. You can try to tune λ by cross validation. For transition probabilities, a better way is to use smoothing techniques we learned in n-gram language models. (Note that the tag sequence is essentially modeled by a bigram language model: $\prod_{i=1}^{n} p(y_i \mid y_{i-1})$.) For example, using interpolation we have

  $$p(y_i \mid y_{i-1}) = \lambda \hat{p}(y_i \mid y_{i-1}) + (1 - \lambda)\hat{p}(y_i) ,$$

  where $\hat{p}$ are the unsmoothed MLE estimates.

- You can use a higher-order HMM to consider wider tag context, e.g. use a trigram model for the tag sequence:

  $$\prod_{i=1}^{n} p(y_i \mid y_{i-2}, y_{i-1}) .$$

  This will require changes to the Viterbi algorithm. Each cell in the lattice now represents a tag bigram, so the runtime becomes $O(n|\mathcal{Y}|^3)$. Read J&M 8.4.7 for more details.

- You may have noticed that the accuracy on novel words is much lower than the accuracy on known words. The strongest source of information for guessing the part-of-speech of unknown words is morphology, e.g. word suffix. Maybe the word "actuary" is a noun because it ends in "ry", or because "actuaries" appears in training data as a noun. Read J&M 8.4.9 for more details on how to build more interesting emission models.

The state-of-the-art performance is around 97%. Note that the performance may not be improved even if you implemented all the above techniques (correctly) because you only have around 100k words for training. Also there is often a trade-off between accuracy and speed. Before you start to implement a technique, do some error analysis to figure out where the performance falls short, e.g. should you improve accuracy on open-class words or closed-class words, which component is taking the most of the time.

4 Grading

We will run your program `tag.py` on a hidden test set and compute accuracy based on `test-output` produced by your program. The file names must match. We will also evaluate the speed of your tagger in wall clock time, which counts both training and inference time. To get full credit (5 points), your accuracy should beat the baseline unigram tagger. The top 25% submissions in terms of accuracy will get 1 extra point. The top 25% submissions in terms of speed will also get 1 extra point.